Zadanie č. 2
Autor:Tomáš Meravý Murárik

Môj projekt je jednoduchý interaktívny program, ktorý sa správa podobne ako shell. Program opakovane čaká na zadanie príkazu od používateľa a následne ho spracuje. Funguje samostatne, ale aj v režime klient-server, ak sú použité prepínače `-c` a `-s`.

V programe je možné zadať nielen port pomocou prepínača `-p`, ale aj IP adresu prichádzajúceho spojenia cez prepínač `-i`. Prepínač `-v` zapína pomocné výpisy na štandardný chybový výstup. S prepínačom `-f` je dostupná funkcionalita tzv. „neinteraktívneho“ shellu, kde sa príkazy spracúvajú zo zadaného súboru. Prepínač `-l` umožňuje zápis logov do určeného súboru. Prepínač `-x` slúži na zadanie jednorazového príkazu.

Splnené boli všetky hlavné úlohy a nasledovné bonusové body: 1, 7, 12, 13, 14, 15 (s úpravou, že program prijíma príkazy cez prepínač `-x` namiesto `-c`, pretože `-x` je v tomto prípade menej mätúci), 18 a 23.

Termín odovzdávania:19.4.2025
Ročník, ak. rok, semester, odbor:2. Ročník, 2. ak. rok, 2. semester, odbor Informatika

```c
#include <arpa/inet.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdarg.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/select.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <time.h>
#include <unistd.h> // name

#define MAX_CONNECTIONS 5
int PORT = 3000;
int server_fd;
int TIMEOUT = 10;
int VERBOSE = 0;
char *IP = NULL;

void help();
int run_server();
int run_client();

void debug_print(const char *format, ...) {
  // function for task -v
  if (!VERBOSE)
    return;

  va_list args;
  va_start(args, format);
  vfprintf(stderr, format, args);
  va_end(args);
}
int change_dir_call(char **line_args) {
  // needed for proper functionality of program
  if (line_args[1] == NULL) {
    perror("missing second argument");
  } else {
    if (chdir(line_args[1]) != 0) {
      perror("error while changing dir");
    }
  }

  return 1;
}
void execute_child(char **line_args) { //  kinder
  char *input = NULL, *output = NULL;
  char **clean_args = malloc(128 * sizeof(char *));
  int idx = 0;
  // checking if something needs contains > or <
  for (int i = 0; line_args[i]; i++) {
    if (strcmp(line_args[i], "<") == 0) {
      input = line_args[++i];
    } else if (strcmp(line_args[i], ">") == 0) {
      output = line_args[++i];
    } else if (strcmp(line_args[i], "") != 0) {
      clean_args[idx++] = line_args[i];
    }
  }
  // if program does contain < or > I give that arg priority
  clean_args[idx] = NULL;
  if (input) {
    int fd = open(input, O_RDONLY);
    if (fd == -1) {
      perror("open");
      exit(EXIT_FAILURE);
    }
    dup2(fd, STDIN_FILENO);
    close(fd);
  } else if (output) {
    int fd = open(output, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    if (fd == -1) {
      perror("open");
      exit(EXIT_FAILURE);
    }
    dup2(fd, STDOUT_FILENO);
    close(fd);
  }
  execvp(clean_args[0], clean_args);
  free(clean_args);
  perror("execvp");
  exit(EXIT_FAILURE);
}

int external_call(char **line_args) {
  // function that activates whenever we call something not
defined in our
  // program
  debug_print("[DEBUG] Executing command: %s\n", line_args[0]);
  pid_t pid;
  // splits into 2
  pid = fork();
  if (pid > 0) {
    // eltern
    waitpid(pid, NULL, 0);
  } else if (pid == 0) {
    // child gets executed
    execute_child(line_args);
  }
  return 1;
```

```
}
int execute_exit_call() {
  // function that quits program
  printf("halt");
  fflush(stdout);
  kill(getppid(), SIGUSR1);
  exit(1);
}
void handle_shutdown(int sig) {
  // in case we need
  if (server_fd > 0)
    close(server_fd);

  kill(-getpid(), SIGTERM);

  exit(0);
}

int execute_args(char **line_args) {
  // main function that handles acts differently based on input
commands given
  if (strcmp(line_args[0], "cd") == 0) {
    change_dir_call(line_args);
  } else if (strcmp(line_args[0], "help") == 0) {
    help();

  } else if (strcmp(line_args[0], "halt") == 0) {
    execute_exit_call();

  } else if (strcmp(line_args[0], "quit") == 0) {
    return 0;
  } else {
    external_call(line_args);
  }

  return 1;
}

char **devide_line(char *line_read, int *index, char *delimiter) {
  // splits line based on some delimiter
  char *not_token;
  char **not_tokens = (char **)malloc(128 * sizeof(char *));

  if (!not_tokens) {
    perror("allocation error");
    exit(1);
  }
  not_token = strtok(line_read, delimiter);
  *index = 0;
  while (not_token != NULL && *index < 31) {
    not_tokens[*index] = not_token;
    (*index)++;
    not_token = strtok(NULL, delimiter);
  }
  not_tokens[*index] = NULL;
  return not_tokens;
}

char *read_line() {
  // receives user input
  char *buffer = NULL;
  size_t bufsize = 0;
  if (getline(&buffer, &bufsize, stdin) == -1) {
    perror("getline");
  }
  buffer[strcspn(buffer, "\n")] = '\0';
  return buffer;
}
char *remove_comment(char *line_read) {
  // removes commends using strchr
  char *pos = strchr(line_read, '#');
  char *new_line;
  if (pos != NULL) {
    new_line =
        (char *)malloc((pos - line_read + 1) * sizeof(char)); //
+1 for '\0'
    if (new_line == NULL) {
      perror("Memory allocation failed");
      exit(1);
    }
    strncpy(new_line, line_read, pos - line_read);
    // add \0 at the end bcs strings
    new_line[pos - line_read] = '\0';
  } else {
    new_line = strdup(line_read);
  }

  return new_line;
}
int main_loop(int client_socket) {
  // main forloop for receiving commands
  char *name = getlogin();
  int status = 1;
  fd_set read_fds;
  struct timeval timeout;

  while (status) {
    time_t now = time(NULL);
    struct tm *timeinfo = localtime(&now);
    printf("%s %d:%d>", name, timeinfo->tm_hour, timeinfo-
>tm_min);
    fflush(stdout);

    FD_ZERO(&read_fds);
    FD_SET(STDIN_FILENO, &read_fds);
    // remember time to keep track of timeout
    timeout.tv_sec = TIMEOUT;
    timeout.tv_usec = 0;

    int activity = select(STDIN_FILENO + 1, &read_fds, NULL, NULL,
&timeout);
    if (activity < 0) {
      perror("select");
      break;
    } else if (activity == 0) {
      // in case of no activity from user close his port
      printf("\nNo input for %d seconds. Closing connection.\n",
TIMEOUT);
      break;
    }
    // receive line
    char *line_read = read_line();
    if (!line_read)
      break;
    // adjust the line
    line_read = remove_comment(line_read);
    int num_commands;
    char **commands = devide_line(line_read, &num_commands, ";");
    // execute every part devided by ; separately
    for (int i = 0; i < num_commands; i++) {
      int num_args;
      char **args = devide_line(commands[i], &num_args, " ");
      status = execute_args(args);
      free(args);
    }

    free(commands);
    free(line_read);
  }

  close(client_socket);
  exit(1);
}
void help() {
  printf("Autor:Tomáš Meravý Murárik\nThis program should be used
as "
         "normal shell\ncommands: cd,ls,help,quit,halt\n\t-
s\tstarts program "
         "in server mode\n\t-c\tstarts program in client mode\n\t-
p "
         "[num]\tworking port to work on\n\t-t [num]\ttime to "
         "timeout\t\t(6b)\n\t-v\tallow debug error printing\t(1b)
\n\t-l "
         "[file]\twrites logs into a file (2b)\t\n\t-f\treads
commands from "
```

```
         "file\t(2b)\n\t-x [arg]\texecutes program given to it as
"
         "arg.(substitude for -c [arg] (2b)\n\t-i [ip]\twill only
accpe "
         "connections from IP (2b)");
}
void execute_script(const char *filename) {
  // code for -f part
  FILE *script = fopen(filename, "r");
  if (!script) {
    perror("fopen");
    return;
  }

  char line[1024];
  while (fgets(line, sizeof(line), script)) {
    line[strcspn(line, "\n")] = '\0';

    if (VERBOSE)
      debug_print("[DEBUG] Processing script command: %s\n",
line);
    int num_args;
    char **args = devide_line(line, &num_args, " ");
    execute_args(args);
    free(args);
  }
  fclose(script);
}

int main(int argc, char *argv[]) {
  int (*run_func)() = &main_loop;
  // read args from user
  for (int i = 1; i < argc; i++) {
    if (argc > 1) {
      if (strcmp(argv[i], "-h") == 0) {
        help();
        return 1;
      } else if (i + 1 < argc && strcmp(argv[i], "-p") == 0) {
        PORT = atoi(argv[i + 1]);
        i++;
      } else if (i + 1 < argc && strcmp(argv[i], "-t") == 0) {
        TIMEOUT = atoi(argv[i + 1]);
        i++;
      } else if (strcmp(argv[i], "-v") == 0) {
        VERBOSE = 1;
      } else if (i + 1 < argc && strcmp(argv[i], "-f") == 0) {
        execute_script(argv[i + 1]);
        exit(1);
      } else if (strcmp(argv[i], "-l") == 0 && i + 1 < argc) {
        freopen(argv[i + 1], "a", stdout);
      } else if (strcmp(argv[i], "-i") == 0 && i + 1 < argc) {
        IP = argv[i + 1]; // Store the IP as a STRING
        i++;
      } else if (strcmp(argv[i], "-x") == 0 && i + 1 < argc) {
        char *cmd = argv[i + 1];
        // run single command
        int sock = socket(AF_INET, SOCK_STREAM, 0);
        struct sockaddr_in serv_addr = {.sin_family = AF_INET,
                                        .sin_port = htons(PORT)};
        inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr);

        if (connect(sock, (struct sockaddr *)&serv_addr,
sizeof(serv_addr))) {
          perror("Connection failed");
          exit(1);
        }

        // Send command
        write(sock, cmd, strlen(cmd));
        write(sock, "\n", 1);

        // Read response
        char buffer[1024];
        ssize_t n;
        while ((n = read(sock, buffer, sizeof(buffer) - 1)) > 0) {
          buffer[n] = '\0';
          printf("%s", buffer);
        }
      }

      if (strcmp(argv[i], "-s") == 0) {
        run_func = run_server;
      } else if (strcmp(argv[i], "-c") == 0) {
        run_func = run_client;
      }
    }
  }
  run_func();
  return 1;
}

void sigchld_handler(int signo) {
  while (waitpid(-1, NULL, WNOHANG) > 0)
    ;
}

int run_server() {
  int new_socket;
  struct sockaddr_in address;
  int opt = 1;
  int addrlen = sizeof(address);

  // create socket
  if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0) {
    perror("socket failed");
    exit(EXIT_FAILURE);
  }

  // set up socket
  if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR |
SO_REUSEPORT, &opt,
                 sizeof(opt))) {
    perror("setsockopt");
    exit(EXIT_FAILURE);
  }

  address.sin_family = AF_INET;

  if (IP) {
    if (inet_pton(AF_INET, IP, &address.sin_addr) <= 0) {
      perror("invalid IP address");
      exit(EXIT_FAILURE);
    }
  } else {
    address.sin_addr.s_addr = INADDR_ANY;
  }
  address.sin_port = htons(PORT);

  if (bind(server_fd, (struct sockaddr *)&address,
sizeof(address)) < 0) {
    perror("bind failed");
    exit(EXIT_FAILURE);
  }

  // start it
  if (listen(server_fd, MAX_CONNECTIONS) < 0) {
    perror("listen");
    exit(EXIT_FAILURE);
  }

  printf("Server running on port %d\n", PORT);

  struct sigaction sa_shutdown;
  sa_shutdown.sa_handler = handle_shutdown;
  sigemptyset(&sa_shutdown.sa_mask);
  sa_shutdown.sa_flags = 0;
  sigaction(SIGUSR1, &sa_shutdown, NULL);

  while (1) {
    if ((new_socket = accept(server_fd, (struct sockaddr
*)&address,
                             (socklen_t *)&addrlen)) < 0) {
      perror("accept");
      continue;
    }
    debug_print("[DEBUG] New connection (socket %d)\n",
new_socket);
    pid_t pid = fork();
    if (pid < 0) {
      perror("fork failed");
      close(new_socket);
    } else if (pid == 0) {
      close(server_fd);

      dup2(new_socket, STDIN_FILENO);
      dup2(new_socket, STDOUT_FILENO);
      close(new_socket);
      main_loop(new_socket);
      close(new_socket);
      exit(EXIT_SUCCESS);
    } else {
      close(new_socket);
    }
  }

  close(server_fd);
  return EXIT_SUCCESS;
}
int run_client() {
  int sock = 0;
  struct sockaddr_in serv_addr;
  if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
    perror("socket creation error");
    return EXIT_FAILURE;
  }
  serv_addr.sin_family = AF_INET;
  serv_addr.sin_port = htons(PORT);

  // set up ip
  if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0) {
    perror("invalid address");
    return EXIT_FAILURE;
  }
  if (connect(sock, (struct sockaddr *)&serv_addr,
sizeof(serv_addr)) < 0) {
    perror("connection failed");
    return EXIT_FAILURE;
  }
  printf("Connected to server\n");

  pid_t pid = fork();
  if (pid == 0) {
    char *buffer = NULL;
    size_t bufsize = 0;
    while (1) {
      if (getline(&buffer, &bufsize, stdin) != -1) {
        if (write(sock, buffer, strlen(buffer)) == -1) {
          perror("write");
          close(sock);
          free(buffer);
          exit(EXIT_FAILURE);
        }
        if (strcmp(buffer, "halt\n") == 0) {
          close(sock);
          free(buffer);
          exit(0);
        }
      }
    }
  } else {
    char resp[1024];
    while (1) {
      ssize_t n = read(sock, resp, sizeof(resp) - 1);
      if (n > 0) {
        resp[n] = '\0';
        if (strcmp("halt", resp) == 0) {
          kill(pid, SIGTERM);
          close(sock);
          exit(1);
        }
        printf("%s", resp);
        fflush(stdout);
      } else if (n == 0) {
        printf("\nServer closed connection\n");
        kill(pid, SIGTERM);
        close(sock);
        exit(EXIT_SUCCESS);
      } else {
        perror("read");
        close(sock);
        exit(EXIT_FAILURE);
      }
    }
  }
  close(sock);
  return EXIT_SUCCESS;
}
```

Zhodnotenie:
Program je funkčný a spĺňa nielen hlavné, ale aj viaceré bonusové požiadavky. Bol vypracovaný pre prostredie Linux. Pre tento program platia viaceré obmedzenia, ako napríklad veľkosť vstupného príkazu, veľkosť vstupného súboru a iné. Všetky tieto obmedzenia by však mali byť nad rámec bežného používania podobného typu programu.

Možných vylepšení pre tento program je viacero. Napríklad, momentálne je používanie znakov „<“ a „>“ obmedzené na jeden výskyt v príkaze, no bolo by vhodnejšie, ak by program vedel spracovať aj viacnásobné použitie týchto znakov v jednom príkaze. Ďalším možným vylepšením by bola napríklad možnosť prispôsobenia promptu, ktorý používateľ vidí, alebo možnosť prijímania príkazov priamo na serveri počas hostovania iných používateľov.

Použité zdroje boli výhradne tie, ktoré boli priložené k zadaniu ako odporúčané materiály.